FUNDAMENTALS OF STACK FILTERS AND THEIR USE IN IMAGE PROCESSING

A Thesis
Presented to
The Academic Faculty

by

Mohit Mendiratta

In Partial Fulfillment
of the Requirements for the
Research Option in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
May 2011

FUNDAMENTALS OF STACK FILTERS AND THEIR USE IN IMAGE PROCESSING

Approved by:

Dr. Edward J. Coyle, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. David Anderson
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved:

ACKNOWLEDGEMENTS

I wish to thank Dr. Edward J. Coyle, Arbutus Chair for the Integration of Research and Education, without whom this project would be impossible. Thank you for taking the time and effort to pursue such a challenging and interesting project with an undergraduate student.

TABLE OF CONTENTS

LIST OF FIGURES

SUMMARY

This thesis is to serve undergraduates attending Georgia Institute of Technology as a starting point for projects related to stack filters. It is written with its intended audience in mind: undergraduate electrical and computer engineers. Most of the mathematical properties are either described fully or referenced to IEEE papers that can easily be obtained.

Image processing is a field that has seen tremendous growth in the last few decades. After computers became common for commercial use in the late 1970s, digital image processing was recognized as an important electrical engineering discipline. Today, there are numerous applications of image processing, such as object recognition, compression, noise reduction, enhancement, etc. The earliest algorithms used in image processing involved linear systems and procedures. These systems have shown good results for certain application such as edge detection; however, linear methods are not robust enough to solve many image processing problems. Rank-Order based algorithms, such as median and stack filters, provide much better results than linear filters in many applications because of their robustness to outliers. Due to the need for rank-order based filters, algorithms must be developed to design filters that are optimized for the problem at hand. Thus, our goal is to find fast algorithms for designing optimal stack filters.

CHAPTER 1

INTRODUCTION

Literature Review

In the last few decades, image processing has seen much change and growth. The algorithms that were considered to be groundbreaking just a few decades ago are now the foundation that every electrical engineering student learns in signal processing courses. Most signal processing students begin their learning with linear methods and filters [1]. For example, they are often taught the use of the moving average filters, which are also known as mean filters, in such common signal processing applications as noise reduction and image recovery [2]. However, towards the end of an introductory signal processing course, students realize that most real world image processing problems cannot be solved using only linear filters because they can very easily be affected by outliers.

The moving average, or mean filter, is the most common and widely used linear filter. Its analog in the class of rank-order based filters is the median filter. There are many variations of the median filter that have proven useful: recursive medians, weighted medians, permutated medians, etc. [3]. One of the common characteristics of these filters is their robustness to outliers. Take, for example, a one-dimensional, integer-valued signal with an outlier. If this signal is to be filtered using a mean filter, the outlier will greatly affect the output – a copy of the filter's impulse response, scaled by the height of the outlier, will appear in the output. If instead the same signal is filtered using a standard median filter, the outlier would be removed and the rest of the output signal would not be affected. An example of such a situation is when an image is corrupted with salt and pepper noise [3]. Such an example is provided in Figure 1 [4]. There are many techniques

in image filtering, each with its own advantages. One of the easiest ways to check the properties of a filter is to run it on a black and white image, which is simply a matrix of integers.

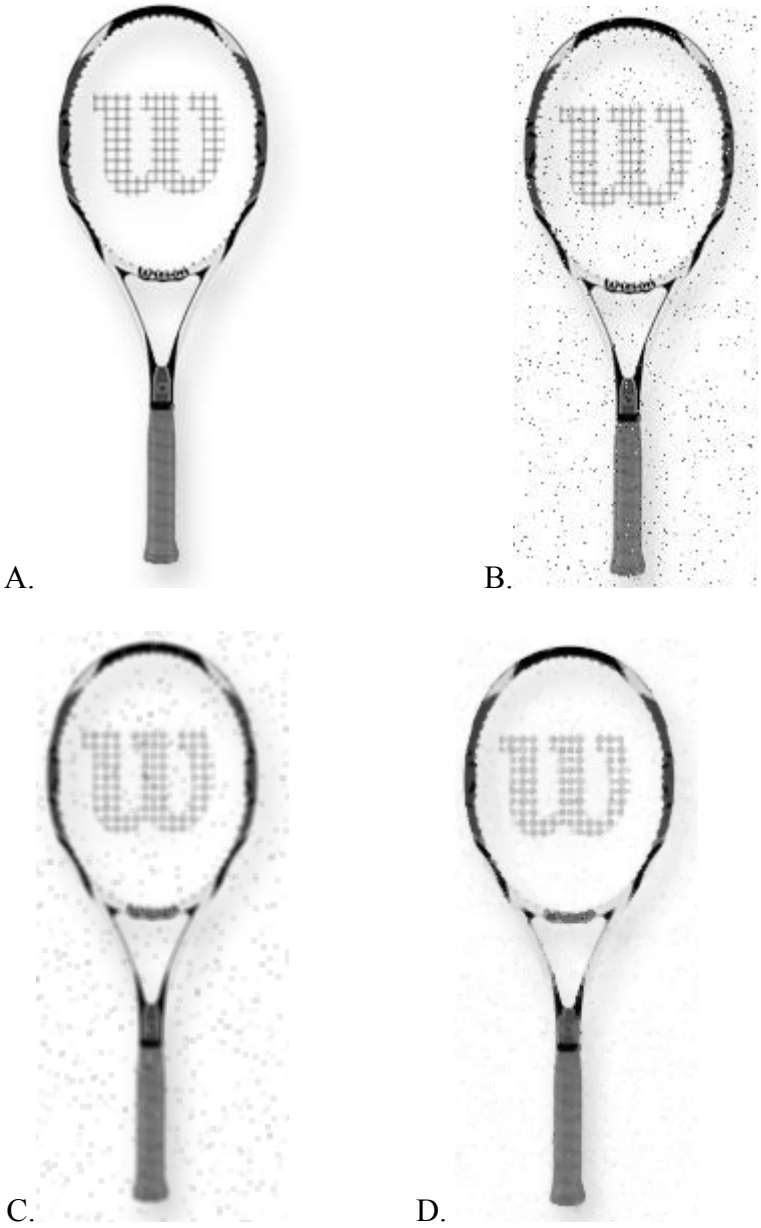

Figure 1. A) Original image, no noise; B) Original image corrupted by salt and pepper noise; C) Result of filtering B with 3x3 mean filter. D) Result of filtering B with 3x3 median filter.

Figure 1 shows that even though the mean filter reduces the level of noise significantly, it also blurs the image. The median filter on the other hand, removes noise while preserving the racket's edges and strings. The median filter can be further improved by using recursive medians or changing the window's size and shapes.

Although it is easy to tell whether one wants to use the mean or median filter in a specific situation, it becomes much more difficult to decide on the filter type as the image complexity grows and filter options increase. For example, the number of stack filters, which are rank-order type filters, for window width n is approximately $2^{2^{\wedge}(n/2)}$. Also, there is a choice of recursive, non-recursive, permutated, weighted, etc. for each operator. Therefore, as the window width increases, it becomes super-exponentially harder to decide on the correct filter to use. Prior research conducted in this area has shown good results [5]. However, further improvement is needed because it is still difficult to design filters with windows that have more than 25 pixels.

Research Problem and its Importance

There are many types of rank order filters, each of which works well in different applications. Choosing which filter to use for a given application can be a difficult decision. For example, there are many situations in which the noise corrupting the input signal is unknown or there are structures in the signal that must be preserved by the filter. Prior research on this problem included work on finding an optimal stack filter using a structural approach, an estimation approach, or a combination of these [5]. The goal of this thesis is to study the theory of optimal stack filtering and see how to improve or expand upon it.

Our goal is to find a theory for optimal stack filtering for a given situation under the mean absolute error criterion. One of the ways to choose a rank order type filter is by using the weighted median filter. In such a filter, the user can define weights for every location of the window. Then, when the weights are applied to the signal, the median is calculated. The application of weights is done by replicating each position in the original signal the number of times specified by the weights. This process can be used to generalize all rank-order filters for window length N. Also, many different choices of weights provide the same filter operator. You can picture this by trying to map different filter types to the truth table of an N bit binary operator. We will try to find an algorithm to calculate the equivalent weight classes for rank order type filters. Our approach will consider both down-sampling and thresholding.

Having a theory to reliably and quickly pick the best rank-order type filter has an enormous number of uses. It could be implemented in many signal processing devices such as cameras, computers, DSP chips, etc. Although there are algorithms already in place for choosing an appropriate rank order filter, they get extremely complex as the window length of the filter increases. A method for finding equivalent weight classes could help in optimizing programs for quicker and more reliable performance.

CHAPTER 2

BACKGROUND

Definitions

All terminology required to understand the rest of the document will be discussed in this chapter. This will serve as a quick tutorial on non linear filtering concepts and some of their mathematical properties. We will begin by defining filters using Boolean operators.

Boolean Operator

All stack filters, which will be defined later, can be written as positive Boolean functions. A positive Boolean operator means that it can be written in the sum of products form without using any complements. As an example, consider a vector [3 7 2]. If we defined a filter with the equation $x_1*x_2 + x_2*x_3 + x_1*x_3$, where * represents the AND operator and + represents the OR operator, then the output would be $x_1 = 3$. This is actually the definition of a median filter for window width 3.

The term 'window width' is a crucial feature of the filter. If the input to a filter is a infinite chain of integers, then a window width 3 filter will consider only 3 integers at a time and perform the Boolean operation on these 3 inputs. As an example, consider an input vector of finite length [3 6 2 8 5 1]. When this is inputted to a median filter, it will be observed as the following sequence:

[3 3 6], [3 6 2], [6 2 8], [2 8 5], [8 5 1], [5 1 1]

The median will be found in each window, and will be placed in the output vector. The first and last windows show a method of handling edges. Here we just duplicate the first and last vector element so that the filter will always have 3 inputs. This also allows the preservation of edges when using a median filter. This process is called a 'sliding window.'

Threshold Decomposition

The Boolean operators only function if the input signal is a Boolean vector. However, as we have seen in the previous section, we must be able to find medians of integer vectors. It is obvious for a human to be able to calculate the median of a sample of 3 integers; however the computer requires a technique called threshold decomposition.

Threshold decomposition begins by converting each input integer into its binary equivalent. Then the integers must be 'stacked' vertically and each level filtered individually. Then the output of each level is combined to make one output vector. It is easiest to explain this using a numerical example.

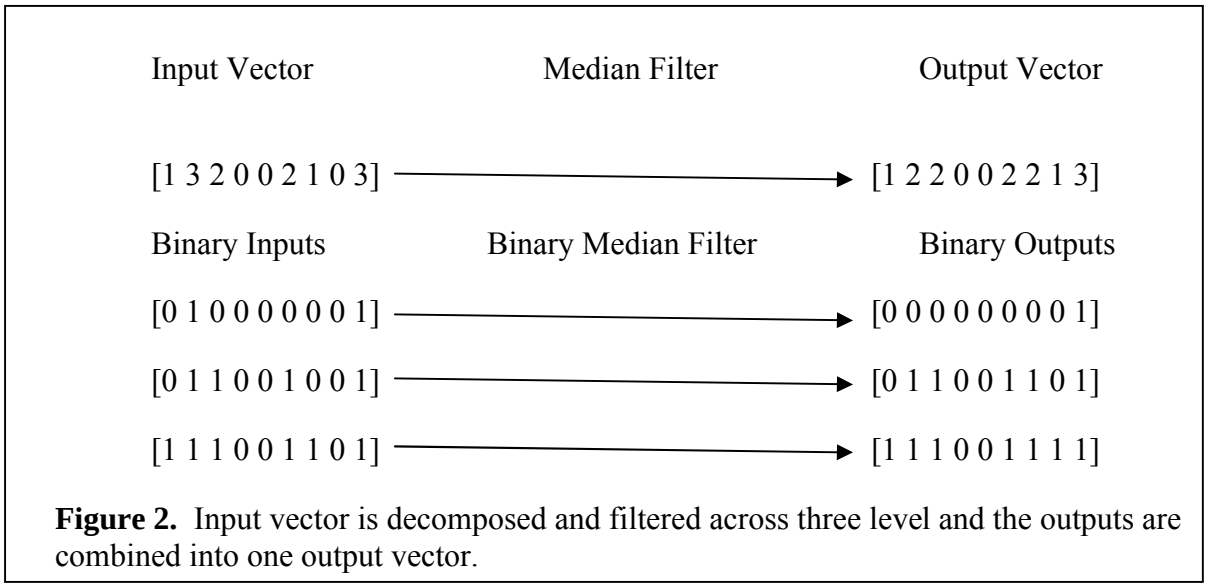

Figure 2. Input vector is decomposed and filtered across three level and the outputs are combined into one output vector.

In the above figure, the input vector is decomposed into its 3 binary vectors. Each of these 3 vectors is said have a level. The integer 2 will have a binary 1 in level 1 and level 2, while having a 0 in level 3. The integer 3 will have a binary 1 in all of the levels 1, 2, and 3. The number of levels required is determined by the largest value in the input vector.

The threshold decomposition property states that the summing each column of binary outputs at a given time instant yields the integer value at that time instant obtained by applying the median directly to the integer-valued input. The threshold decomposition is thus a type of superposition property. It is not a general superposition property, like the one that defines linear filters; it requires the input signal is broken up into binary signals by thresholding, as described above. The existence of this property is the key to the later theory of minimum mean absolute error stack filtering. It is also a surprising parallel for median filters to the superposition property for linear filters [5].

The threshold decomposition leads to the second property of stack filters, called the stacking property.

Stacking Property

The stacking property says that each value in a binary level, starting from 1, must be greater than or equal to the value in same location in next level. If the input vector is $X_1 \ldots X_n$, then the threshold decomposition will create S levels according to the largest integer value in the input. The stacking property says that $S_1 \geq S_2 \geq \ldots \geq S_{end}$ for every location in S [5]. This can be seen in the previous example were the integer 2 in location 3 has the value of 1 in S_1 and S_2 while having a value of 0 in S_3.

The threshold decomposition and stacking properties combined with positive Boolean functions completely define stack filters. This means that for a window width 3 filter, there are 20 positive Boolean operators, implying there are 20 window width 3 stack filters.

Rank Order Operators

Rank order operators are a sub class of stack filters that are very well suited for image processing applications. The common minimum, maximum, and median filters are all rank order operators. The number of rank order filters depends on the window width. A window width 3 filter has 3 rank order filters; the 1st rank order filter, more commonly known as the minimum, the 2nd rank order filter or median, and the 3rd rank order filter or maximum. The number of rank order operators is the same as the window width

One way to visualize these rank order filters is by first arranging the window in ascending order. Then, each rank is just the position in the window. Figure 1 shows this concept pictorially:

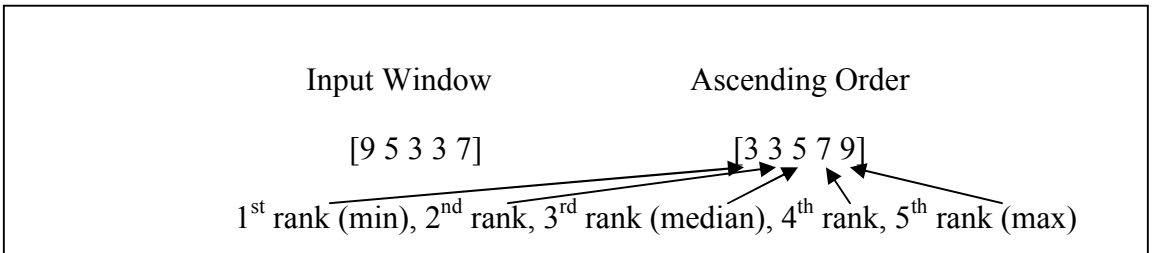

Figure 3. Definition of rank in a window width 5 filter.

Due to the enormous number of stack filters available even for small window widths, we will focus mainly on the rank order sub class. Rank order filters often prove to be the most useful filters, along with a select few stack filters, for noise reduction.

CHAPTER 3

GRAPHICAL USER INTERNFACE

One of our goals is to design and simulate non-linear filters so we can get a better understanding of their behavior. In addition reading IEEE journals and books on nonlinear signal processing, we chose to create graphical user interfaces (GUIs) so that future students will be able to quickly learn the characteristics of non linear filters [6].

Types of GUIs

We created several GUIs in which the user can enter 1-D signals and then choose from over 16 filters to generate the output. The output signal could then be filtered repeatedly until the root signal is obtained [8]. An example of such a GUI is shown below.

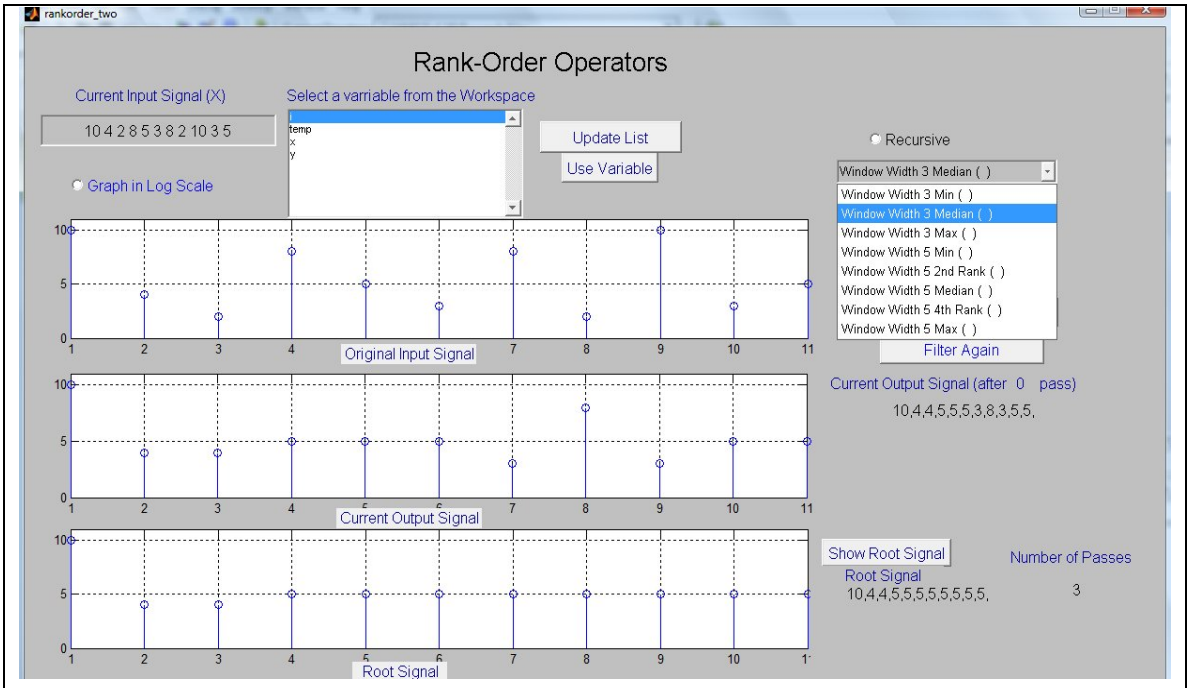

Figure 4. Screenshot of a GUI created in MATLAB that implements many different rank order operators.

GUI's such as these will be used in class demonstrations to show students the characteristics of nonlinear filters, such as their robustness to outliers. The graphs of the input and output signal show where the outliers have been removed after filtering. Also, it makes it easy for students to check whether they understand the functionality of each filter. We also created GUI's that take in a noisy image and output a figure after it has been filtered by a separable median, rectangular median, or weighted media. The image filtering GUI can show students the trade-off between noise reduction and detail preservation. Along with these GUIs, we have created programs for edge detection using threshold decomposition. These applications have helped me develop a good understanding of non linear filters and their use in image processing.

1-D GUI

We created many 1-D GUIs starting from a median filter, where the user can choose an arbitrary window width and input a signal. Next, we created a GUI that allowed the use of recursive median filters of arbitrary window width, chosen by the user. Next, we incorporated the use of other rank order filters. Due to the complexity and number of these filters, we only designed these for window width 3 and 5. All of these GUIs were then combined into one, having all of the important features from the previous versions.

Main Features of the Final Rank Order Operations 1-D GUI

There are two methods to enter the input signal in this GUI. The first allows the user to specify a vector in the “Current Input Signal (X)” box. This is useful for smaller signals as it gives the user complete control of each point. The second method is to specify a vector in Matlab in the current workspace and import it into the GUI. This

feature is very useful for long input vectors or when the input vector is generated from another Matlab script. Importing the vector to the GUI is very simple; the user just needs to click the "Update List" button, and the vector will appear in the list box.

Once the input is chosen, the "Original Input Signal" graph will display the signal. The graph axes can be changed to log scale by selecting the "Graph in Log Scale" button. This is useful in case the input signal's dynamic range is large. Once the user chooses to filter the signal, the output for each filter pass will be displayed in the "Current Output Signal" graph. If the user clicks on the "Show Root Signal" button, the last graph will show the root signal and display the numerical values in vector form to the right of the graph. All of the three windows have the same scale and size so that comparisons can be made between them.

The choice of the filter is made using the drop-down menu in the top right hand side of the GUI. There are 8 filter choices, each of which can be recursive or non recursive. Once the desired filter is chosen, the input signal can be filtered by clicking the "Filter Again" button. The resultant output is displayed on the graph and in vector form just under the filter button. The number of "passes" or times the original signal is filtered is also updated. This allows the user to observe how the original filter approaches the root signal.

The last important feature in the 1-D GUI is saving the output back to the current workspace. This can be done by clicking the "Save Output to Workspace" button. This saves the current output vector into the current workspace with the variable name "rank_order_out." This is very convenient if the user would like to use the output vector into another Matlab script.

2-D GUI

The 2-D GUI is used for taking the input image, corrupted with noise, and outputting a filtered image to observe the performance of the filter. The input image can be of any size, however once it becomes larger than 640x640, the filtering can take a significant amount of time (greater than 10 seconds.) The input image is converted to black and white, and is displayed for the user. Once the user selects the desired filter, the output image is displayed in another window. This allows the user to compare the input and output side by side. The windows are not scaled in anyway, so there is no distortion in the images. The original image dimensions are always maintained. This is very important because scaling an image can introduce noise and corrupt the image, making the comparison of the input and output files difficult. Below is the design of the 2-D GUI.

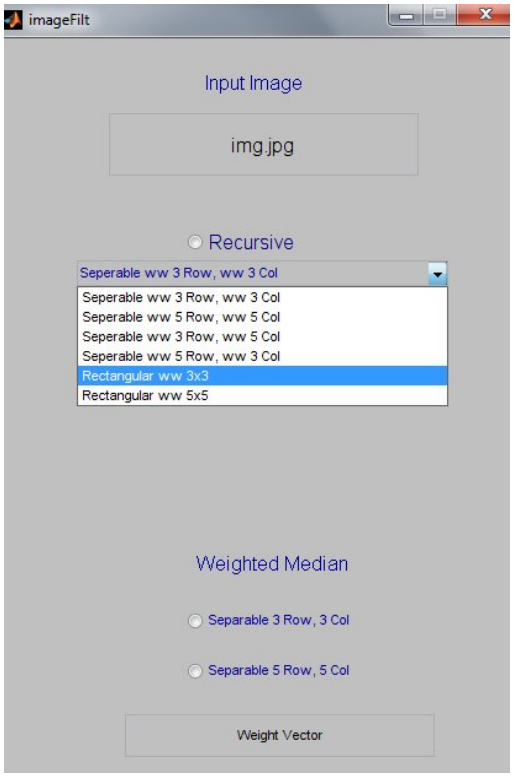

Figure 5. Screenshot of the 2-D image filtering GUI *imageFilt*.

Filter Choices

There are three types of filter choices: separable, rectangular and weighted median filters. Each of the three filters provides different results, which are described in Chapter 5, and have different applications.

Separable Median Filter

The separable median filter is the easiest 2-D filter to understand because it is very similar to the 1-D median filters. The separable median filter is defined as a 1-D filter in each of the X and Y directions. If the input image to be filtered is a 200 by 200 pixel image, the computer representation will be a 200 by 200 integer matrix. A window width 3 by 5 separable median filter would first filter all of the rows with a window width 3 median filter. This would output a 200 by 200 row filtered matrix. This filtered matrix is then filtered column by column using a window width 5 median filter. If this is difficult to grasp, picture transposing the matrix and then filtering each row. This is effectively filtering the columns, just like in the previous 1-D cases. The final output will be a 200 by 200 image matrix filtered by a window width 3 by 5 separable median filter.

Weighted Separable Median Filter

The weighted separable median filter functions in a very similar fashion to the regular separable median filter. It first filters the rows and then the columns. The difference comes into play with the weighting vector [7]. It is easiest to understand the filter in a 1-D case. Consider the case with an input vector [12 6 4 1 9] and a weight vector [1 2 3 2 1]. The following diagram illustrates the weighted median operator:

Input(Xn) = [12 6 4 1 9]

Weight Vec = [1 2 3 2 1]

Output = Median[1 ◊ 12, 2 ◊ 6, 3 ◊ 4, 2 ◊ 1, 9 ◊ 1]

Output = Median[12, 6, 6, 4, 4, 4, 1, 1, 9]

Output = Median[1, 1, 4, 4, 4, 6, 6, 9, 12]

Output = 4.

Figure 6. Step by step procedure showing the functionality of a weighted median filter.

The result of the weighted median filter was the output of 4, while a regular median would have outputted 6. In the 2-D case, the same principles apply.

Rectangular Median Filter

The rectangular median filter is a 'true' 2-D filter because it does not have a 1-D equivalent. It is easiest to understand the rectangular median filter using images. Figure 7 describes the rectangular median filter. The input image is broken into a 3 by 3 matrix. The center point in the matrix, $X_{2,2}$ is the median output.

0	6	4	5	7	6	9	0	4
5	0	7	7	9	9	9	3	6
1	3	3	4	3	4	5	9	9
7	5	8	3	8	9	9	6	5
6	2	9	8	8	4	9	9	2
5	2	6	9	3	0	7	8	0
6	6	0	0	6	9	2	7	1
8	6	8	5	2	8	5	1	4
6	9	2	7	1	8	4	6	4

Input Image - InputMatrix

0	6	4
5	0	7
1	3	3

3 Rectangular filter at $InputMatrix_{2,2}$

0	0	1
3	3	4
5	6	7

Ordered 3 by 3 Window around $InputMatrix_{2,2}$

Figure 7. Shows the output of a rectangular median filter on the input matrix at location 2,2.

The median is found to be 3. This result is different from the separable median because the rectangular median takes into account the neighbors in both directions, not just the X or Y directions.

Recursive

Each of the filter types described above, separable, rectangular and weighted, can be modified by being implemented recursively. This is same principle as described in Chapter 2, where as the window ‘slides’ the outputs from the previous operations affect the output of the next operation.

Adding Noise (Corruption)

The addition of noise or other corruptions is a very useful way of testing filter responses. In Matlab this can be done quite easily with the *imnoise( )* in-built function. This function allows the addition of Gaussian, Poisson, Salt & Pepper, and many other types of signal corruption. One clear advantage of this function is that it allows the user to take any clean image, which would have certain properties like edges and curves, and observe the transformations to these properties and its ability to remove unwanted noise.

CHAPTER 4

STACK FILTERS

Chapter 2 provided the basic mathematical properties of stack filters, namely threshold decomposition and the stacking property. However, these properties are not enough to actually generate a list of all stack filters for a given window width. It was said earlier that even though we will focus on rank order operators, there are still some important stack filters that are not a part of the rank order sub class. This chapter will focus on deriving all positive Boolean functions for a given window width, which is equivalent to finding all stack filters for a given window width.

Mathematical Procedure for Finding All Stack Filters for a Given Window Width

There are 7581 stack filters for window width 5, and more approximately 2^{35} stack filters of window width 7 [5]. This means that even though for window width 3 it may be possible to directly determine which if the 20 stack filters is best in an application, we need more powerful techniques for filters with larger windows. The first thing we need is a model of how the input signal evolves in the filter's window. Therefore, we model the input vectors and noise as Markov chains and define their state transition matrices. The transition matrices are manipulated according to the window width of the desired stack filter into a single state transition matrix. Then a cost vector is chosen to find the optimal stack filter. The dimensions of all of these matrices grow exponentially with window width.

Definition of Input Signal and Noise

The original signal is modeled as an irreducible, discrete time Markov chain, I(t) for t=0,1,…. Due to the threshold decomposition and stacking properties, we will consider only Boolean signals. This allows us to model the original vector via a state transition matrix that defines how the binary sequences in the filter change as the filter window moves across the signal. Our goal is to find the state transition probabilities that form the entries of the state transition matrix that defines this Markov Chain, I(t). We will call this matrix P_S.

The noise to be applied can be modeled in the same fashion, using a state transition matrix, P_N. It is important to note that P_S and P_N do not need to be of the same dimensions, as the noise and original vector will most likely have a different number of states. The input to the stack filter will be a function of the original signal and noise vector. We have a great deal of freedom in defining how the signal and noise interact with each other.

To stay consistent with terminology used across this topic, we will define the received process to be the possible combinations of signal and noise that are then used to define the actual signal that the filter sees via some functions. The state space of the received process is the product of these signal and noise state spaces. Since we assume the signal and noise are independent, the state transition matrix of the received process is $P_R = P_S \otimes P_N$, where $\otimes$ denotes the Kronecker product.

The entries of are just the products of the probabilities of each possible combination of possible states for the input and noise signals. The stack filter will still only observe a sequence of binary bits that are determined by the binary function applied

to each state in the state space of the received process. To model what the stack filter will observe, based on the window size, we construct a state transition matrix of the window process, P_W [5].

$$P_W = \sum_{i=1}^{m} \overline{\mathbf{1}_{m,i}} \otimes (\overline{\mathbf{1}_{m,i}^T} * \boldsymbol{P}_R) \otimes \overline{\mathbf{1}_{m,i}^T} \otimes \boldsymbol{I}_{m^{b-2} \times m^{b-2}} \otimes \overline{\mathbf{1}_m}$$

where $\overline{\mathbf{1}_{m,i}}$ is a column vector with 1 in the i'th entry

$\boldsymbol{I}_{m^{b-2} \times m^{b-2}}$ is a m^{b-2} by m^{b-2} identity matrix

$\mathbf{1}_m$ is a column vector of m 1's

T represents the transpose operator

Finding the Optimal Stack Filter

Finding the optimal binary stack filter now requires a cost function. This cost function is defined as a vector that assigns values to each output of the filter. For this project, the cost for the filter output to be correct (meaning the output at time t is the same as the input without noise corruption at time t) as 0. The cost for an incorrect output is 1.

Lastly, to find the optimal stack filter, we solve the linear programming problem using P_W and the cost vector. For this project, we will use the in-built Matlab function *linprog* for solving the linear program.

Explanation of Matlab Scripts

We created a Matlab function called *stackfilter( )*. This takes the input as a window width that is any integer up to 9. If the input is greater than 9, Matlab will run out of memory and the script will be forced to terminate. The function implements the above section using a random vector of weights every time it is run. The functionality was checked by observing outputs for window width 3.

We also created another Matlab script that functions as a numerical example for the section above. It was called *phasematrix.m*.

Explanation of *phasematrix.m*

First, the input and noise transition matrices are defined. These are chosen for to be 4x4 and 3x3 matrices for relative numerical simplicity. Next, the received process state transition matrix, P_W, is created. This matrix is 1728x1728 is size. The complexity of the matrix already makes it difficult to find the limiting probabilities for each window vector.

The limiting probabilities are the left eigenvectors with eigenvalue of 1 for P_W. These are stored in a vector called *piebar*. This vector is used to define the cost vector. It is very difficult to obtain an algorithm for any input and noise transition matrix to finding the cost vector. Therefore, the next part of the code was found by manually drawing P_W and observing which rows and columns reflect specific costs. These costs are then outputted into the workspace. These costs can be used with the *stackfilter( )* algorithm to find the optimal stack filter.

Our results were for *phasemarix.m* were confirmed by comparing them with the results obtained by Moncef Gabbouj, a former graduate student of Dr. Edward Coyle.

CHAPTER 5

RESULTS AND CONCLUSIONS

We were able to complete ourr low target of creating GUI's to be used in non linear signal processing classes. This was our main goal when the project began in the summer of 2008. The rank order operators and 2 two image filtering GUIs have been submitted to Dr. James McClellan and may be used in the introductory signal processing course, ECE 2025. This part of the project was completed in early 2009.

Our medium target was to create a Matlab script that allows us to define stack filters of large window widths, up to 11. We were able to generate the stack filters for window widths up to 9. However, Matlab does not have enough virtual memory to support our algorithm for window width 11. We also used random weights for the linear program and were able to confirm the functionality of the script by observing outputs for window width 3. Due to requirement of understanding linear programming and other statistical concepts not taught in the undergraduate electrical engineering curriculum, a large amount of time was spent on building mathematical background. This was finished in Spring 2009.

We were not able to extend our algorithm past window width 9 and therefore our high goal was not accomplished. However, any undergraduate or graduate student is welcome to review the work done in this project, and to continue and try to accomplish our high target. This thesis is meant as a guide to any future student interested in finding theories of optimal stack filtering.

APPENDIX A

MATLAB CODE FOR RANK-ORDER OPERATORS GUI

This GUI allows the instructor to quickly show some of the key properties of the nonrecursive and recursive versions of rank order filters of window widths 3 and 5.The signal to be filtered can be entered directly in the Input window or the signals stored in the workspace can be used by loading them and then selecting the desired one. Clicking on the "Update List" button will show all of these signals.

There are three graphs: (1) The Input signal; (2) The Output signal that results from applying the selected filter to the Input signal; and (3) The root signal that results from repeated filtering of the Input with a Nonrecursive filter or from one pass of a Recursive filter. These three graphs can be plotted on a log scale (base 10) if the signal range is too large to be shown conveniently in a non-log scale.

A simple way to demonstrate a filter would be to enter a signal in the "Input" box, and then click on the 'Filter to Root' button. This will show you the root signal and the number of passes required by the nonrecursive version of that filter to produce it. Next, you can click on the 'Filter' button and progressively see the signal changed into the root. Note: if you change the filter type while repeatedly filtering a signal, it will take as input the last filtered output, not your original input; the root would then be different.

The signal currently displayed in the "Output" box can be stored in the current workspace at any time by clicking on the 'Save Output to Workspace' button. It is saved

as 'rank_order_out'. Each time this button is clicked, the previous value of rank_order_out is overwritten.

Pre-Defined Signals:

There are four pre-defined signals that can be seen once you click on the 'Update List' button. These can be used to show the signal structures that are removed or preserved by these filters. The guiding principles for the median filters are:

(a) signals that are monotone non-decreasing or non-increasing in every window position of the filter are root signals;

(b) impulses that are less than 1/2 the window width of the filter are removed; step edges are passed without blurring.

1) "Impulses-Edges" is a signal with step-edges and positive- and negative-going impulses of widths 1 and 2. The window width three median filter will remove the impulses of width 1 but leave those of width 2. The window width five median filter will remove impulses of width 1 and 2. Both filters will preserve the step-edges. Note that the height of the impulses and edges makes no difference. Application: Suppressing impulsive noise in images. The median removes salt and pepper noise completely; any linear filter would leave a copy of its impulse response at the location of the impulse. Edges, which are common in images, are left unchanged; a linear filter would blur any step edge.

2) "Oscillation" is a signal that oscillates with period 2 between the values 3 and -3. It starts with 3 and ends with -3, which causes the recursive and nonrecursive versions of the median filter to produce different root signals.

Application: Associative Memory Behavior. Every finite length signal is filtered to a root signal after a finite number of passes of the median filter, and after one pass by a recursive median filter. A signal is a root of a median filter if and only if it is a root of the recursive median. Note that a given input signal, like the one in this example, may be filtered to different roots by the recursive and nonrecursive medians.

3) "WW3-Root" and "WW5-Root" are signals that roots of the window width 3 and window width five median filters, respectively. Note that the WW5 root is a root of the WW3 median filter but the WW3 root is *not* a root of the WW5 median filter. Thus, the set of roots of the window width 5 median is a subset of the set of roots of the window width 3 median.

Application: The Structural Approach to Designing Median Filters. In general, a median filter of window width 2K+1 removes signal structures of widths K or smaller and preserves those of width K+1 or larger. This provides useful guidance when making an initial selection of a filter width for a filtering task.

```
function varargout = fast_rank_order(varargin)

% This filter shows the result when an input signal of integers that is
filtered through a rank
% operator of window width 3 or 5. For ww3, rank 1 is min, rank 2 is
median,
% and rank 3 is the max. For ww5, its the min, 2nd order, median, 4th
order
% and max.
% The useer is allowed to import a vector of any length from the
worksapce
```

```
% and use it in the GUI. The variable should be of type double.
%
% There are 3 embedded functions: get_var_names, fast_rank_op, and
% find_root. get_var_names gets variables from the workspace into the
GUI.
% fast_rank_op filters the signals according input_signal,
window_width,
% order, and rec/non-recursive. find_root is a function that calls
% fast_rank_op and finds the root signal.

% Last Modified by GUIDE v2.5 12-Jan-2009 20:36:51
% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @fast_rank_order_OpeningFcn, ...
                   'gui_OutputFcn',  @fast_rank_order_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT

% --- Executes just before fast_rank_order is made visible.
function fast_rank_order_OpeningFcn(hObject, eventdata, handles,
varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to fast_rank_order (see VARARGIN)

% Choose default command line output for fast_rank_order
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes fast_rank_order wait for user response (see UIRESUME)
% uiwait(handles.figure1);

% --- Outputs from this function are returned to the command line.
function varargout = fast_rank_order_OutputFcn(hObject, eventdata,
handles)
varargout{1} = handles.output;
function input_signal_Callback(hObject, eventdata, handles)

a=get(handles.input_signal,'String');
```

```
b=str2num(a);
    if (~get(handles.log_scale,'Value'))   %not log scale
        axes(handles.input_graph);
        stem(b);
        axis([0 length(b)+1 -(max(abs(b))+1) (max(abs(b))+1)]);
        grid on;
    else %log scale
        axes(handles.input_graph);
        stem(real(log10(b)));%real part if input in negative...
        ylabel('Log Scale');
        grid on;
    end
set(handles.filter_pass_text,'String',0);

function input_signal_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in update_push.
function update_push_Callback(hObject, eventdata, handles)

signal1 = [1 1 4 1 1 12 12 4 4 4 -9 -4 -4 -4 7 7 7 7 -3 -3 1 1 2 2 3 4
6 7 9 9 9 9 ];
assignin('base','Impulses_Edges',signal1);

signal2 = [3 -3 3 -3 3 -3 3 -3 3 -3 3 -3 3 -3 3 -3 3 -3 3 -3 3 -3 3 -
3];
assignin('base','Oscillation',signal2);

signal3 = [1 1 3 3 2 1 -2 -5 -5 4 4 4 6 6 6 6 -2 -2 -2 -2 9 9 -8 -8 2 4
5 7 7 -7 -7 7 7 -7 -7 7 7];
assignin('base','WW3_Median_Root',signal3);

signal4 = [1 1 3 3 2 1 -2 -5 -5 -3 0 3 6 6 6 -2 -2 9 9 9 7 5 3 -1 -5 -5
3 3 3 7 7 7 -7 -7 -7 7 7 7];
assignin('base','WW5_Median_Root',signal4);

vars = evalin('base','who');
set(handles.variable_box,'String',vars);

function variable_push_Callback(hObject, eventdata, handles)

a=get_var_name(handles);
b=evalin('base',a);
    if (~get(handles.log_scale,'Value'))   %not log scale
        axes(handles.input_graph);
        stem(b);
        axis([0 length(b)+1 -(max(abs(b))+1) (max(abs(b))+1)]);
        grid on;
    else %log scale
        axes(handles.input_graph);
```

```
        stem(real(log10(b)));%real part if input in negative...
        ylabel('Log Scale');
        grid on;
    end
set(handles.filter_pass_text,'String',0);
b=num2str(b);
set(handles.input_signal,'String',b);
% --- Executes on button press in recursive.
function recursive_Callback(hObject, eventdata, handles)
% hObject    handle to recursive (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of recursive

% --- Executes on selection change in variable_box.
function variable_box_Callback(hObject, eventdata, handles)
% hObject    handle to variable_box (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = get(hObject,'String') returns variable_box contents
as cell array
%        contents{get(hObject,'Value')} returns selected item from
variable_box


% --- Executes during object creation, after setting all properties.
function variable_box_CreateFcn(hObject, eventdata, handles)
% hObject    handle to variable_box (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: listbox controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in popupmenu1.
function popupmenu1_Callback(hObject, eventdata, handles)

% --- Executes during object creation, after setting all properties.
function popupmenu1_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

```
% --- Executes on selection change in popupmenu2.
function popupmenu2_Callback(hObject, eventdata, handles)

% --- Executes during object creation, after setting all properties.
function popupmenu2_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes on button press in save_out.
function save_out_Callback(hObject, eventdata, handles)

a=get(handles.output_text, 'String');
out=str2num(a);
assignin('base','rank_order_out',out); %assings var out to the
workspace
% --- Executes on button press in filter.

function filter_Callback(hObject, eventdata, handles)
f=str2num(get(handles.filter_pass_text,'String'));
    a=get(handles.input_signal,'String');
    b=str2num(a);
c=get(handles.popupmenu1,'Value');
if (~get(handles.recursive,'Value'))%non-recursive operations
    switch c
        case 1
            d=fast_rank_op(b,3,1,1);
        case 2
            d=fast_rank_op(b,3,2,1);
        case 3
            d=fast_rank_op(b,3,3,1);
        case 4
            d=fast_rank_op(b,5,1,1);
        case 5
            d=fast_rank_op(b,5,2,1);
        case 6
            d=fast_rank_op(b,5,3,1);
        case 7
            d=fast_rank_op(b,5,4,1);
        case 8
            d=fast_rank_op(b,5,5,1);
    end
else%if recursive
        switch c
        case 1
            d=fast_rank_op(b,3,1,2);
        case 2
            d=fast_rank_op(b,3,2,2);
        case 3
            d=fast_rank_op(b,3,3,2);
        case 4
            d=fast_rank_op(b,5,1,2);
        case 5
            d=fast_rank_op(b,5,2,2);
```

```
        case 6
            d=fast_rank_op(b,5,3,2);
        case 7
            d=fast_rank_op(b,5,4,2);
        case 8
            d=fast_rank_op(b,5,5,2);
    end
end

if (~get(handles.log_scale,'Value'))   %not log scale
    axes(handles.output_graph);
    stem(d);
    axis([0 length(b)+1 -(max(abs(b))+1) (max(abs(b))+1)]);
    grid on;
else %log scale
    axes(handles.output_graph);
    stem(real(log10(d)));%real part if input in negative...
    ylabel('Log Scale');
    grid on;
end
e=num2str(d);
set(handles.output_text,'String',e);%show output string
f=f+1;
f=num2str(f);
set(handles.filter_pass_text,'String',f);%update number of passes
d=num2str(d);
set(handles.input_signal,'String',d);%update filtered string

function output_text_Callback(hObject, eventdata, handles)

% --- Executes during object creation, after setting all properties.
function output_text_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes on button press in log_scale.
function log_scale_Callback(hObject, eventdata, handles)

function edit3_Callback(hObject, eventdata, handles)

function edit3_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes on button press in root_button.
function root_button_Callback(hObject, eventdata, handles)
    a=get(handles.input_signal,'String');
    b=str2num(a);
c=get(handles.popupmenu1,'Value');
```

```
if (~get(handles.recursive,'Value'))%non-recursive operations
    switch c
        case 1
            [r,p]=find_root(b,3,1,1);
        case 2
            [r,p]=find_root(b,3,2,1);
        case 3
            [r,p]=find_root(b,3,3,1);
        case 4
            [r,p]=find_root(b,5,1,1);
        case 5
            [r,p]=find_root(b,5,2,1);
        case 6
            [r,p]=find_root(b,5,3,1);
        case 7
            [r,p]=find_root(b,5,4,1);
        case 8
            [r,p]=find_root(b,5,5,1);
    end
else%if recursive
        switch c
        case 1
            [r,p]=find_root(b,3,1,2);
        case 2
            [r,p]=find_root(b,3,2,2);
        case 3
            [r,p]=find_root(b,3,3,2);
        case 4
            [r,p]=find_root(b,5,1,2);
        case 5
            [r,p]=find_root(b,5,2,2);
        case 6
            [r,p]=find_root(b,5,3,2);
        case 7
            [r,p]=find_root(b,5,4,2);
        case 8
            [r,p]=find_root(b,5,5,2);
    end
end

if (~get(handles.log_scale,'Value'))   %not log scale
    axes(handles.root_graph);
    stem(r);
    axis([0 length(b)+1 -(max(abs(b))+1) (max(abs(b))+1)]);
    grid on;
else %log scale
    axes(handles.root_graph);
    stem(real(log10(r)));%real part if input in negative...
    ylabel('Log Scale');
    grid on;
end
r=num2str(r);
set(handles.root_text,'String',r);
p=num2str(p);
set(handles.root_num_text,'String',p);
```

```
function var1 = get_var_name(handles)
% Returns the names of the two variables to plot
list_entries = get(handles.variable_box,'String');
index_selected = get(handles.variable_box,'Value');
var1 = list_entries{index_selected(1)};

function filtered_signal =
fast_rank_op(input_signal,window_width,order,rec)
% performs rank order operations on input_signal according to window
width
% and order. If rec is equal 1, then the operations are non recursive.
If
% rec is anything but 1, the operations are recursive.

type = rem(window_width,2); %if the windowlength is even, the answer is
0

if type == 0
    append = window_width/2;
else
    append = (window_width-1)/2;
end

arr=[];
for i=1:length(input_signal)
    arr(i+append)=input_signal(i);
end
for i=1:append
    arr(i)=input_signal(1);
    arr(end+1)=input_signal(end);
end
% arr is the properly appended array
if (rec==1)%non recursive function
    for i=1:length(input_signal)
    temp=sort(arr(i:i+window_width-1));
    filtered_signal(i)=temp(order);
    end
else %recursive
    for i=1:length(input_signal)
    temp=sort(arr(i:i+window_width-1));
    filtered_signal(i)=temp(order);
    arr(i+append)=filtered_signal(i);
    end
end

function [root,passes]=find_root(input_signal,window_width,order,rec)
x={[9999]};
x{2}=[input_signal];
j=2;
while j<10000
    if x{j}==x{j-1}
        break;
    else

filtered_signal=fast_rank_op(input_signal,window_width,order,rec);
        x{1+j}=filtered_signal;
```

```
        input_signal=filtered_signal;
    end
    j=j+1;
end
root=filtered_signal;
passes=length(x)-3;
```

APPENDIX B

MATLAB CODE FOR 2-D *IMAGEFILT* GUI

```
function varargout = imageFilt(varargin)
% IMAGEFILT M-file for imageFilt.figf
%      IMAGEFILT, by itself, creates a new IMAGEFILT or raises the
existing
%      singleton*.
%
%      H = IMAGEFILT returns the handle to a new IMAGEFILT or the
handle to
%      the existing singleton*.
%
%      IMAGEFILT('CALLBACK',hObject,eventData,handles,...) calls the
local
%      function named CALLBACK in IMAGEFILT.M with the given input
arguments.
%
%      IMAGEFILT('Property','Value',...) creates a new IMAGEFILT or
raises the
%      existing singleton*.  Starting from the left, property value
pairs are
%      applied to the GUI before imageFilt_OpeningFunction gets called.
An
%      unrecognized property name or invalid value makes property
application
%      stop.  All inputs are passed to imageFilt_OpeningFcn via
varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only
one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help imageFilt

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @imageFilt_OpeningFcn, ...
                   'gui_OutputFcn',  @imageFilt_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
```

```
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT

end
% --- Executes just before imageFilt is made visible.
function imageFilt_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to imageFilt (see VARARGIN)

% Choose default command line output for imageFilt
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes imageFilt wait for user response (see UIRESUME)
% uiwait(handles.figure1);

end
% --- Outputs from this function are returned to the command line.
function varargout = imageFilt_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;
end
function inputimage_Callback(hObject, eventdata, handles)
% hObject    handle to inputimage (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of inputimage as text
%        str2double(get(hObject,'String')) returns contents of
inputimage as a double
x=get(handles.inputimage,'String');
im=imread(get(handles.inputimage,'String'));
figure
imshow(im(:,:,1));
end
% --- Executes during object creation, after setting all properties.
function inputimage_CreateFcn(hObject, eventdata, handles)
% hObject    handle to inputimage (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
```

```
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
end

% --- Executes on selection change in popupmenu1.
function popupmenu1_Callback(hObject, eventdata, handles)
% hObject    handle to popupmenu1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = get(hObject,'String') returns popupmenu1 contents
as cell array
%        contents{get(hObject,'Value')} returns selected item from
popupmenu1

 im=imread(get(handles.inputimage,'String'));
 [row col]=size(im(:,:,1));

if (~get(handles.togglebutton1,'Value'))
    switch get(handles.popupmenu1,'Value')

        case 1 %separable 3R, 3C
        output=[];
        newout=[];
           for i=1:row
%               filter each row
                 output=[output
                        oneDFilter(im(i,:,1),3)];
           end
           for j=1:col
               newout=[newout
                        oneDFilter(output(:,j,1),3)];
           end
           figure
           imshow(newout',[0 255])
        case 2 %separable 5R, 5C

                output=[];
        newout=[];
           for i=1:row
%               filter each row
                 output=[output
                        oneDFilter(im(i,:,1),5)];
           end
           for j=1:col
               newout=[newout
                        oneDFilter(output(:,j,1),5)];
           end

         figure
           imshow(newout',[0 255])

        case 3 %separable 3R, 5C
```

```
        output=[];
        newout=[];
           for i=1:row
%               filter each row
                output=[output
                       oneDFilter(im(i,:,1),3)];
           end
           for j=1:col
               newout=[newout
                       oneDFilter(output(:,j,1),5)];
           end
        figure
           imshow(newout',[0 255])

        case 4 %separable 5R, 3C
         output=[];
         newout=[];
           for i=1:row
%               filter each row
                output=[output
                       oneDFilter(im(i,:,1),5)];
           end
           for j=1:col
               newout=[newout
                       oneDFilter(output(:,j,1),3)];
           end

         figure
           imshow(newout',[0 255])
       case 5 %rectangular 3x3
          output=ordfilt2(im(:,:,1),5,ones(3,3),'symmetric');
    figure
           imshow(output,[0 255])
       case 6 %rectangular 5x5
          output=ordfilt2(im(:,:,1),13,ones(5,5),'symmetric');
 figure
           imshow(output,[0 255])



    end
else
    switch get(handles.popupmenu1,'Value')

        case 1 %separable 3R, 3C
        output=[];
        newout=[];
           for i=1:row
%               filter each row
                output=[output
                       recMed(im(i,:,1),3)];
           end
           for j=1:col
               newout=[newout
                       recMed(output(:,j,1),3)];
           end
```

```
  figure
           imshow(newout',[0 255])
         case 2 %separable 5R, 5C

                 output=[];
         newout=[];
            for i=1:row
%                filter each row
                  output=[output
                         recMed(im(i,:,1),5)];
            end
            for j=1:col
                newout=[newout
                         recMed(output(:,j,1),5)];
            end
 figure
           imshow(newout',[0 255])
         case 3 %separable 3R, 5C
         output=[];
         newout=[];
            for i=1:row
%                filter each row
                  output=[output
                         recMed(im(i,:,1),3)];
            end
            for j=1:col
                newout=[newout
                         recMed(output(:,j,1),5)];
            end
     figure
           imshow(newout',[0 255])
         case 4 %separable 5R, 3C
          output=[];
          newout=[];
            for i=1:row
%                filter each row
                  output=[output
                         recMed(im(i,:,1),5)];
            end
            for j=1:col
                newout=[newout
                         recMed(output(:,j,1),3)];
            end
  figure
           imshow(newout',[0 255])
       case 5 %rectangular 3x3
          output=recRectMed(im,3);
         figure
         imshow(output, [0 255]);
       case 6 %rectangular 5x5
          output=recRectMed(im,5);
         figure
         imshow(output,[0 255]);

    end
end
```

```
end
% --- Executes during object creation, after setting all properties.
function popupmenu1_CreateFcn(hObject, eventdata, handles)
% hObject    handle to popupmenu1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
end
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
% %
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
function y = oneDFilter(x,n)
% inputs: array to be filtered -> x
%         rank of filter (window length) -> n
%
% this function will append the indices will the start and ending
values.
% this will make sure that the edges do not get blurred.

type = rem(n,2); %if the windowlength is even, the answer is 0

if type == 0
    append = n/2;
else
    append = (n-1)/2;
end
append;
   arr = zeros([1 length(x)+2*append]);
for i=1:append
%     arr = zeros([1 length(x)+2*append]);
    arr(i)=x(1);        %create an array to be filtered with appended
    arr(i+length(x)+append)=x(end);    %beginnings and ends
end

for i=1:length(x)
    arr(i+append)=x(i); % copies values of x into arr while keeping
end                     % the edges
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
count=1;
count2=n+1;
firstarr=arr(1:n);
sortedfirstarr=sort(firstarr);
y(1)=sortedfirstarr(append+1);
for i=2:(length(x)-2)
    b=1;
    while ~(arr(count)==sortedfirstarr(b))
            %remove oldest element
           b=b+1;
    end
```

```
    sortedfirstarr(b)=[];

    sortedfirstarr;
    for c=1:(n-1)
        if (arr(count2)>=sortedfirstarr(end)) %largest
            sortedfirstarr(end+1)=arr(count2);
            sortedfirstarr;
            break;
%              in-between
        elseif
(arr(count2)>=sortedfirstarr(c))&&(arr(count2)<=sortedfirstarr(c+1))
            halfarr=sortedfirstarr(1:c);
            halfarr(end+1)=arr(count2);
            sortedfirstarr=[halfarr sortedfirstarr((c+1):end)];
            sortedfirstarr;
            break;

%              smallest
        elseif (arr(count2)<=sortedfirstarr(1))
            half=sortedfirstarr;
            sortedfirstarr=[arr(count2) half];

            break;
        end
    end



    y(i)=sortedfirstarr(append+1);
    count=count+1;
    count2=count2+1;

end
    %last 2 must be done manually
        sortedfirstarr=sort(arr((count2-(n-1)):(end-1)));
        y(i+1)=sortedfirstarr(append+1);
        sortedfirstarr=sort(arr((count2-(n-2)):end));
        y(i+2)=sortedfirstarr(append+1);

end
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
% %
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
% %
function y = WMF(x,n,wvec)
type = rem(n,2); %if the windowlength is even, the answer is 0

tempvec=[];
% normalize the vector, but doesnt change wvec for primes
for i=1:length(wvec)
    if ~isprime(wvec)
        tempvec=wvec./sum(wvec);
        temp=min(wvec);
        tempvec=wvec./temp;
```

```
    else
        tempvec=wvec;
        break;
    end
end
wvec=tempvec;

if type == 0
    append = n/2;
else
    append = (n-1)/2;
end
append;
   arr = zeros([1 length(x)+2*append]);
for i=1:append
%     arr = zeros([1 length(x)+2*append]);
    arr(i)=x(1);        %create an array to be filtered with appended
    arr(i+length(x)+append)=x(end);    %beginnings and ends
end

for i=1:length(x)
    arr(i+append)=x(i); % copies values of x into arr while keeping the
edges
end
newarr=[];

for i=1:length(x)
    temp=[];
    newarr=arr(i:(n+i-1));
    for j=1:n
        newarr2=[];
        for k=1:wvec(j)
            newarr2=[newarr2 newarr(j)];
        end
   temp=[temp newarr2];
    end
    sorted=sort(temp);
    len=sum(wvec);
    if mod(len,2)==0 %even
        y(i)=(sorted((len/2)+1)+sorted(len/2))/2;
    else
        y(i)=sorted(ceil(len/2));
    end
end

end

function ww_Callback(hObject, eventdata, handles)
% hObject    handle to ww (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of ww as text
%        str2double(get(hObject,'String')) returns contents of ww as a
double
end
```

```
% --- Executes during object creation, after setting all properties.
function ww_CreateFcn(hObject, eventdata, handles)
% hObject    handle to ww (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
end



function weights_Callback(hObject, eventdata, handles)
% hObject    handle to weights (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of weights as text
%        str2double(get(hObject,'String')) returns contents of weights
as a double

 im=imread(get(handles.inputimage,'String'));
 [row col]=size(im(:,:,1));
if get(handles.weight3,'Value')&&(~get(handles.weight5,'Value'))
    wstring=get(handles.weights,'String');
    wvec=str2num(wstring);
    im(1,1,:)=0;
    im(end,end,:)=255;
            output=[];
        newout=[];
           for i=1:row
%               filter each row
                 output=[output
                        WMF(im(i,:,1),3,wvec)];
           end
           for j=1:col
               newout=[newout
                      WMF(output(:,j,1),3,wvec)];
           end
 figure
           imshow(newout',[0 255])

elseif get(handles.weight5,'Value')&&(~get(handles.weight3,'Value'))
    wstring=get(handles.weights,'String');
    wvec=str2num(wstring);
       im(1,1,:)=0;
    im(end,end,:)=255;

            output=[];
        newout=[];
```

```
            for i=1:row
%                filter each row
                  output=[output
                         WMF(im(i,:,1),5,wvec)];
            end
            for j=1:col
                newout=[newout
                         WMF(output(:,j,1),5,wvec)];
            end
 figure
            imshow(newout',[0 255])
else
    errordlg('Please select either WW 3 or WW 5, not both')
end
end

% --- Executes during object creation, after setting all properties.
function weights_CreateFcn(hObject, eventdata, handles)
% hObject    handle to weights (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
end

% --- Executes on button press in weight5.
function weight5_Callback(hObject, eventdata, handles)
% hObject    handle to weight5 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of weight5
n=5;
end

% --- Executes on button press in weight3.
function weight3_Callback(hObject, eventdata, handles)
% hObject    handle to weight3 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of weight3

n=3;
end



% --- Executes on button press in togglebutton1.
function togglebutton1_Callback(hObject, eventdata, handles)
```

```
% hObject    handle to togglebutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of togglebutton1


end


function y = recMed(x,n)
type = rem(n,2); %if the windowlength is even, the answer is 0

if type == 0
    append = n/2;
else
    append = (n-1)/2;
end
arr=[];
arr = zeros([1 length(x)+2*append]);
for i=1:append
   arr(i)=x(1);
   arr(i+length(x)+append)=x(end);
end
for i=1:length(x)
    arr(i+append)=x(i);
end

for i=1:length(x)
    sorted=sort(arr(i:(n-1+i))); % take median of chunks of the array
    y(i)=sorted(ceil(n/2));
    arr(i+1)=y(i);
end
end
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
% %
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
% %
function y = recRectMed(x,n)
% input x is a matrix and n is the window width
vec=x(:,:,1);
type = rem(n,2);
if type == 0
    append = n/2;
else
    append = (n-1)/2;
end

firstrow=vec(1,:);
firstcol=vec(:,1);
lastrow=vec(end,:);
lastcol=vec(:,end);

[row col]=size(vec);
mat=zeros(2*append+row,2*append+col);
```

```
% copy matrix leaving 0s for the rows and cols we need to append
for i=1:row
   for j=1:col
       mat(append+i,append+j)=vec(i,j);
   end
end
if n==3
    for i=1:length(firstrow)
        mat(1,i+1)=firstrow(i);
        mat(end,i+1)=lastrow(i);
    end
    for j=1:length(firstcol)
        mat(j+1,1)=firstcol(j);
        mat(j+1,end)=lastcol(j);
    end
elseif n==5
    for i=1:length(firstrow)
        mat(1,i+2)=firstrow(i);
        mat(2,i+2)=firstrow(i);

        mat(end-1,i+2)=lastrow(i);
        mat(end,i+2)=lastrow(i);
    end
    for j=1:length(firstcol)
        mat(j+1,1)=firstcol(j);
        mat(j+1,2)=firstcol(j);

        mat(j+1,end-1)=lastcol(j);
        mat(j+1,end)=lastcol(j);

    end

else
    errordlg('Sorry, function is only for ww 3 and ww 5')
end
% now the actual filtering with rectangular window
newmat=[];
temp=[];
if n==3
    for i=1:1:row
        for j=1:1:col
            for k=1:1:3
                for h=1:1:3
                    temp=[temp mat(k+i-1,h+j-1)];
                    h;
                end
                k;
            end
       newmat(i,j)=median(temp);
       mat(i,j)=newmat(i,j);
       temp=[];
        end

    end
else
        for i=1:1:row
```

```
        for j=1:1:col
            for k=1:1:5
                for h=1:1:5
                    temp=[temp mat(k+i-1,h+j-1)];
                    h;
                end
                k;
            end
        newmat(i,j)=median(temp);
        mat(i,j)=newmat(i,j);
        temp=[];
         end

    end

end
y=newmat;
end
```

APPENDIX C

MATLAB CODE FOR 2-D *STACKFILTER( )*

```
function ret = stackfilter(n)
W(3)={[1,-1,0,0,0,0,0,0;0,0,1,-1,0,0,0,0;1,0,-1,0,0,0,0,0;0,1,0,-
1,0,0,0,0;0,0,0,0,1,-1,0,0;0,0,0,0,0,0,1,-1;0,0,0,0,1,0,-
1,0;0,0,0,0,0,0,1,0,-1;1,0,0,0,-1,0,0,0;0,1,0,0,0,-1,0,0;0,0,1,0,0,0,-
1,0;0,0,0,1,0,0,0,-1]};
x=n-3; %number of intermediate W matrices before size N
for i=1:x
    [xcor,ycor]=size(W{3+i-1});
    zerot(xcor,ycor)=0;
    Wtemp=W{3+i-1}
    W{3+i}=[Wtemp zerot;
           zerot Wtemp;
           eye(2^(3+i-1)) -eye(2^(3+i-1))];
    W{3+i-1}=[];
end
A=[W{n};
    linspace(0,0,(2^n)-1) 1;
    -1 linspace(0,0,(2^n)-1)];

% picking random weights C
[r,c]=size(A);
b=[linspace(0,0,(r-2)) 1 0];
C = (5*randn(1,2^n));
ret=linprog(C,A,b);
```

APPENDIX D

MATLAB CODE FOR 2-D *PHASEMATRIX.M*

```
clear
clc

% Qs={(0,a),(0,b),(1,a),(1,b)}
% Qn={X,Y,z}

%state transition matrix of original signal
Ps=[0 1 0 0;
    0 .4 .6 0;
    0 0 0 1;
    .4 0 0 .6];
%state trans matrix of noise corruption
Pn=[.6 .2 .2;
    .8 .1 .1;
    .6 .3 .1];
%state trans matrix of received processes seen by stack filter
Pr=kron(Ps,Pn);
Pw=zeros(1728,1728);
m=12;
for i=1:m

onemi=zeros(m,1);
onemi(i,1)=1;

onem=ones(1,m)';
eyem=eye(m,m);
prod=(onemi')*Pr;
t1=kron(onemi,prod);
t2=kron(eyem,onem);
t1=kron(t1,onemi');
t1=kron(t1,t2);

Pw=Pw+t1;
end
sum(Pw');
% piebar is the limiting probability vector
% this is the eigenvector with eignevalue = 1
%
[eigvec,eigval]=eig(Pw.');
eigvec=conj(eigvec);
% eigval=conj(eigval);
% eigvec=abs(eigvec);
% eigval=round(eigval);
size(eigvec);

% % eigen values are on the diagonal of the eigval matrix...
[r,c]=size(eigval);
newr=[];
```

```
newc=[];
for i=1:r
    for j=1:c
        if (eigval(i,j)==1)
            newr=[newr i];
            newc=[newc j];
        end
    end
end

% now define the cost function...

piebar=eigvec(:,1);
piebar=piebar/sum(piebar);
%
aA=0;
for i=1:12:144
    aA=aA+piebar(i,1);
end
aA
aD=0;
for i=290:12:432
    aD=aD+piebar(i,1);
end
aD
aG=0;
for i=290:12:432
    aG=aG+piebar(i,1);
end
aG
eA=0;
for i=435:12:576
    eA=eA+piebar(i,1);
end
eA
eD=0;
for i=580:12:720
    eD=eD+piebar(i,1);
end
eD
eG=0;
for i=725:12:864
    eG=eG+piebar(i,1);
end
eG
iA=0;
for i=870:12:1008
    iA=iA+piebar(i,1);
end
iA
iD=0;
for i=1015:12:1152
    iD=iD+piebar(i,1);
end
iD
iG=0;
```

```
for i=1200:12:1584
    iG=iG+piebar(i,1);
end
iG
mA=0;
for i=1305:12:1440
    mA=mA+piebar(i,1);
end
mA
mD=0;
for i=1450:12:1584
    mD=mD+piebar(i,1);
end
mD
mG=0;
for i=1595:12:1728
    mG=mG+piebar(i,1);
end
mG
```

References

[1] *Signal Processing First: A Multimedia Approach*, J. H. McClellan, R.W. Schafer, M.A. Yoder, Upper Saddle River, NJ: Prentice-Hall, Inc., (1998).

[2] *Exploratory Data Analysis*. Tukey, John W, Addison-Wesley. (1977).

[3] *Nonlinear Signal Processing: A Statistical Approach*, Gonzalo R. Arce, Wiley Inc., 2004

[4] Tennis Warehouse. Image available online: http://www.tennis-warehouse.com/viewlarge.html?PCODE=K61T90 (Accessed: 10/03/2008)

[5] Coyle, E.J.; Lin, J.-H.; Gabbouj, M. "Optimal stack filtering and the estimation and structural approaches to image processing", *Acoustics, Speech and Signal Processing, IEEE Transactions on,* Volume: 37 Issue: 12 Dec 1989

[6] R. Ashi and A. Ameri, "Introduction to Graphical User Interface (GUI) in MATLAB 6.5", IEEE UAEU Student Branch,

[7] Gabbouj, M; Astola, J.; Neuvo, Y. "Optimal weighted median filtering under structural constraints", *Signal Processing, IEEE Transactions on*, Volume: 43 Issue: 3 Mar 1995

[8] Butz, A.R., "Root signals of rank order processors", *Signal Processing, IEEE Transactions on*, Volume: 40 Issue: 1 Jan 1992